Manipulation de partitions et arbres couvrants

le 12 novembre 2024

L'objectif de ce TP est d'implémenter l'algorithme de Kruskal de recherche d'un arbre couvrant de poids minimal. Pour cela, il faut savoir gérer des partitions. Nous commencerons donc par étudier la représentation des partitions et la structure de données (très classique) Union-Find.

1 Partitions et Union-Find

On considère n objets (que nous appellerons $0, 1, \ldots, n-1$) sur lesquels on définit des partitions. L'objectif est de construire une structure permettant d'effectuer les opérations suivantes :

- initialiser à la partition la plus fine où chaque classe est un singleton ;
- déterminer si deux éléments sont dans la même classe;
- modifier la relation pour fusionner les classes de deux éléments ;
- compter le nombre de classes de la partition ;

Remarque 1. *Tout le monde se souvient du "théorème" (c'est plus un changement de vocabulaire qu'un théorème d'ailleurs) vu en math en sup reliant les notions de partitions et de relations (classes) d'équivalence? C'est ce théorème qui justifie le vocabulaire utilisé ici.*

1.1 Une solution naïve

Donner une implémentation en mettant simplement à jour un tableau v tel que `v.(x)` contient l'élément minimal de la classe d'équivalence de x.

▷ **Question 1.** Écrire les fonctions suivantes :

```
initialise :  int -> int array
compte :  int array -> int
test_equ :  int array -> int -> int -> bool
fusion :  int array -> int -> int -> unit.
```

Quelle est la complexité de chacune de ces opérations ? ◁

1.2 Utiliser des arbres et des forêts

On peut représenter une partition par une forêt d'arbres. Par exemple la partition $\{0,1,5\},\{2,3\},\{4\}$ pourra être représentée par l'une des forêts ci-dessous :

On représentera ce genre de forêt par un couple `(taille,pere)`, où `taille` est le nombre de parties de la partition, c'est-à-dire le nombre d'arbres de la forêt, et `pere.(x)` vaut `x` si `x` est la racine de son arbre, et la valeur de son père dans son arbre sinon.

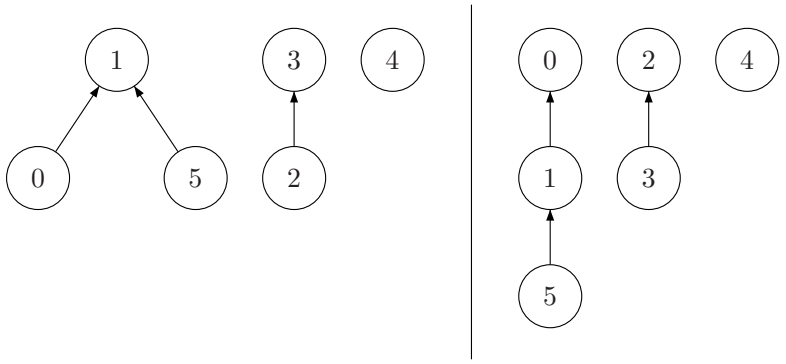
Ainsi la structure utilisée est du type :

```
type structure = { mutable taille :  int ; pere :  int array } ;;
```

▷ **Question 2.** Écrire pour cette nouvelle structure les fonctions suivantes :

```
initialise :  int -> structure
compte :  structure-> int
test_equ :  structure -> int -> int -> bool
fusion :  structure -> int -> int -> unit.
```

Quelle est la complexité de chacune de ces opérations ? ◁



1.3 Optimisation

Une première optimisation, appelée "union par rang", consiste à limiter la profondeur des arbres lorsque l'on effectue une fusion : pour cela, on maintient à jour un nouveau tableau `poids` qui permettra de déterminer lequel des deux arbres à fusionner deviendra un sous arbre de l'autre. Le tableau poids associe à chaque **racine** d'arbre le poids de l'arbre en question et associe une valeur arbitraire ($n+1$ par exemple) à chacun des autres sommets.

La notion de poids "naturelle" à considérer ici est la profondeur de l'arbre. On verra qu'elle n'est pas si pratique car non compatible avec la deuxième optimisation. On prendra donc comme valeur de `poids` la **taille** de l'arbre (i.e. son nombre de sommets).

La structure devient :

```
type structure = { mutable taille :  int ; pere :  int array ; poids :  int array } ;;
```

▷ **Question 3.** Modifier les fonctions pour utiliser la nouvelle structure. ◁

La seconde optimisation est la compression des chemins : on remarque que l'on n'a aucun besoin de connaître le vrai père de chaque élément : il serait bien plus efficace de diriger `père.(x)` sur la racine de l'arbre auquel appartient `x` (ce qui correspondrait à la première implémentation pour laquelle la fusion est trop coûteuse). Ainsi, **à chaque recherche de la racine** (on tire profit du fait qu'on parcourt de toute façon une branche), on mettra à jour les valeurs du tableau père des éléments qu'on passe en revue en les reliant à la racine. On compresse donc les chemins reliant un sommet rencontré à la racine de l'arbre.

▷ **Question 4.** Modifier les fonctions en faisant de la compression des chemins. Que se passe t-il pour le tableau `poids` ? Que se serait-t-il passé si le poids avait été la profondeur de l'arbre? ◁

2 Une première application

On considère un graphe non orienté donné par ses listes d'adjacence, autrement dit représenté par le type :

```
type voisin = int list ;;  type graphe == voisin array;;
```

▷ **Question 5.** Utiliser la structure d'union-find pour rédiger une fonction qui calcule les composantes connexes de ce graphe. `composantes : graphe -> partition` ◁

3 Algorithme de Kruskal

On considère un graphe non orienté G connexe et pondéré à poids positifs. Un arbre couvrant du graphe $G=(V,A)$ est un arbre $Ar=(V,A')$ connexe tel que $A'\subset A$. L'objectif de ce TP est aussi de trouver un arbre couvrant minimal (soit, de poids minimum) de G.

Dans l'algortihme de Kruskal, on utilise une représentation des graphes différentes de celles que l'on a manipulées jusqu'à maintenant : liste des arêtes. Ainsi le type graphe pondéré ici est défini par :

`type graphe = int*int*int list` où le triplet (i,j,p) correspond à l'arête (i,j) pondérée par le poids p.

▷ **Question 6.** écrire une fonction qui détermine le nombre de sommets n d'un graphe connexe à partir de la liste de ses arêtes. ◁

L'algorithme de Kruskal construit un arbre couvrant minimal en procédant de la façon suivante :

1. On trie les arêtes par poids croissant.
2. On considère chaque sommet comme un arbre à un élément.
3. Si une arête relie deux arbres distincts de la forêt, on ajoute cette arête à A et on fusionne les deux arbres, sinon on la jette.

▷ **Question 7.** Implémenter l'algorithme de Kruskal à l'aide d'une structure union-find. Quelle est sa complexité ? ◁

4 Une variante de Kruskal.

Soit $G = (S, A, f)$ un graphe pondéré non orienté connexe et $T = (S, B)$ un arbre couvrant minimal de G. On appelle :

- arête dangereuse une arête qui est de poids maximal dans un cycle de G;
- arête utile une arête qui n'appartient à aucun cycle de G.

On suppose pour les questions théoriques seulement que f est injective.

▷ **Question 8.** Montrer que B ne contient aucune arête dangereuse. ◁

▷ **Question 9.** Montrer que B contient toutes les arêtes utiles. ◁

L'algorithme que nous allons ici considérer suit le principe suivant :
On parcourt les arêtes par ordre décroissant de poids. Si on tombe sur une arête dangereuse, on la supprime. Les questions précédentes garantissent la correction d'un tel algorithme.

▷ **Question 10.** Ecrire une fonction `chemin : graphe->int->int->bool` telle que l'appel à `chemin g x y` détermine s'il existe un chemin entre `x` et `y` dans `g`. On pourra adapter en Ocaml la fonction qui calcule les composantes connexes faite en première partie. ◁

Pour tester si une arête $a = \{s, t\}$ est dangereuse ou non, on crée une copie de G dans laquelle on supprime a et toutes les arêtes de poids strictement supérieur à $f(a)$. S'il existe encore un chemin de s à t, c'est que a est dangereuse.

▷ **Question 11.** Ecrire une fonction `supp_ar : graphe ->int->int->int->void` telle que `supp_ar g x y p` supprime l'arête (x, y, p) dans le graphe `g`. ◁

▷ **Question 12.** Ecrire une fonction `copie : graphe->graphe` qui copie un graphe. ◁

▷ **Question 13.** Ecrire une fonction `dangereuse : graphe->int->int->int->bool` telle que `dangereuse g x y` teste si l'arête (x, y, p) est ou non dangereuse dans `g`. ◁

▷ **Question 14.** Ecrire une fonction `aretes : graphe-> (int*int*int) list` qui renvoie la liste des arêtes du graphe passé en argument. ◁

▷ **Question 15.** Implémenter l'algorithme précédent sous la forme d'une fonction `kruskal2 : graphe -> graphe` qui calcule et renvoie un arbre couvrant minimal. ◁